Fundamenta Informaticae 82 (4): 341–359
IOS Press

Validating Behavioral Component Interfaces in Rewriting Logic *

Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen
Department of Informatics, University of Oslo, Norway
{einarj, olaf, aribraat}@ifi.uio.no

Abstract. Many distributed applications can be understood in terms of components interacting in an open environment such as the Internet. Open environments are subject to change in unpredictable ways, as applications may arrive, evolve, or disappear. In order to validate components in such environments, it can be useful to build simulation environments which reflect this highly unpredictable behavior. This paper considers the validation of components with respect to behavioral interfaces. Behavioral interfaces specify semantic requirements on the observable behavior of components, expressed in an assume-guarantee style. In our approach, a rewriting logic model is transparently extended with the history of all observable communications, and metalevel strategies are used to guide the simulation of environment behavior. Over-specification of the environment is avoided by allowing arbitrary environment behavior within the bounds of the assumption on observable behavior, while the component is validated with respect to the guarantee of the behavioral interface.

Keywords: Validation, components, behavioral interfaces, adaptive testing, rewriting logic

1. Introduction

This paper suggests an application of rewriting logic [25] to test the black-box behavior of software units in *open distributed environments* such as the Internet. An open environment is an environment in which various other software units exist, and little or no information about these units is available. A distributed environment is an environment in which communication is asynchronous. Many critical applications and infrastructures run in open distributed environments; e.g., bank services, air traffic control, online tax forms, and other electronic government services. Reasoning in this setting is intrinsically difficult, partly due to the non-determinism caused by the distribution, but more characteristically due to the unknown and evolving open environment.

*This research is in the context of the EU project IST-33826 *CREDO: Modeling and analysis of evolutionary structures for distributed services* (`http://credo.cwi.nl`).

It is a major challenge to predict the behavior of components evolving in open distributed environments, in order to ensure and maintain behavioral properties concerning safety, availability, quality of service, robustness, and fault tolerance. Formal approaches to system verification, such as Hoare logic, type checking, and model checking, traditionally depend on knowing the implementation details of the system components, including those in the open environment. In contrast, assume-guarantee reasoning systems abstract from the implementation details of the environment when verifying those of a system component [23, 24, 29], thus providing compositional verification techniques. Approaches based on testing simulate an environment in which the system can be subjected to test runs. In this paper, we take an assume-guarantee approach to testing. In contrast to verification methods, testing cannot generally ensure that components are always well-behaved, but testing may still give revealing insights into a component's behavior. However, conformance testing for software units in open distributed environments is challenging [35]. This paper shows how open environments can be mimicked by underspecified formal descriptions based on *observable behavior* in order to validate the black-box behavior of software units in open distributed environments at the modeling level. Model-based testing in the early stages of component design makes the development process more efficient [28].

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [16] and used in, e.g., .Net and Corba. In this paper, we model distributed components as concurrent objects which exchange messages in an asynchronous way. Method calls are mimicked by pairs of initiation and completion messages for each call. The models are executable in the rewriting logic system Maude [8], which has built-in support for simulation, model checking, and verification. However, Maude does not directly support the validation of black-box behavior. In order to allow black-box validation, we use requirement specifications in terms of observable behavior. Observable behavior is specified using *behavioral interfaces* [18, 19] which describe component services available to the environment. In contrast to [18, 19], where behavioral interfaces were used for specification, this paper is oriented towards testing and uses behavioral interfaces to derive test scenarios in an executable setting.

This paper presents an executable framework for validating the observable behavior of models in the open distributed setting. For this purpose, behavioral interfaces are captured in rewriting logic and combined with a standard rewriting logic model of asynchronously communicating objects. Furthermore, the executable platform in Maude is extended with validation facilities for the observable behavior of components in a transparent way. Rewriting logic is *reflective* [7, 9] in a mathematically precise manner: it is possible to reason formally about reflective rewriting inside rewriting logic itself, and to execute reflective specifications at the Maude *metalevel*. The use of reflection is essential to our approach, allowing for guided search and system monitoring in a modular, composable, and hierarchical way. Reflection may be used to define execution strategies for an executable object model; for example, a *non-deterministic* execution strategy is proposed in [20]. Reflective specifications support a layered architecture where several specifications may be given at each level, extending a system model with, e.g., logging facilities [34]. In this paper, we transparently extend, at the metalevel, an executable specification with its history of observable communications and define execution strategies at the metalevel which are guided by semantical requirements on the communication history. One strategy is used to mimic open environments and another to test the executable model. The two strategies are combined in order to enable an assume-guarantee style model-based testing of components with respect to their behavioral interfaces. This paper extends a previous paper by the authors [21]; in particular, the running example is new to this paper.

Paper overview: Section 2 presents a formalism for behavioral interfaces. Section 3 presents rewriting logic and the Maude tool. Section 4 develops metalevel strategies for monitoring and testing

executable Maude models. A strategy for simulation of open environments is presented in Section 5 and it is shown how such a strategy can be applied for testing. Section 6 discusses related and future work.

2. Behavioral Interfaces

An open distributed system (ODS) can be represented by components or objects that run in parallel and communicate asynchronously by means of remote method calls. The implementation details of the components may be unknown, in which case reasoning must rely on abstract specifications of the system's components. We assume that components come equipped with *behavioral interfaces* that instruct us on how to use them. As a component may be used for multiple purposes, it can come equipped with *multiple* interfaces. This section presents a formalism for viewpoints based on a notion of generic interface with behavioral requirements, restricted to safety aspects. For further details about this work, see [18, 19].

Black-box specifications of concurrent components may be expressed in terms of *observable behavior*; i.e., the time sequence of input and output to the components. This fits well with the notion of encapsulation; only visible operations are considered at the specification level. An execution can be represented by a sequence of communication events, which is infinite for non-terminating executions. However, infinite sequences are not easy to reason about. To avoid infinite sequences, specifications may be expressed in terms of the finite initial segments of the executions, capturing the abstract states of the components during execution. These sequences are commonly referred to as histories [10] or traces [15]. Prefix-closed sets of finite traces express safety properties in the sense of Alpern and Schneider [3].

Finite sequences. Consider an abstract data type $\mathsf{Seq}[T]$ of finite sequences parameterized by a type T. Functions over sequences will be defined by means of convergent sets of equations, using the empty sequence, ε, and right append, $\_;\_ : \mathsf{Seq}[T] \times T \to \mathsf{Seq}[T]$, as sequence constructors. The underscore "_" denotes argument positions of functions with mix-fix notation. Let $\mathsf{Set}[T]$ denote the type of sets parameterized by a type T and Nat the natural numbers.

We define projection, $\_/\_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \to \mathsf{Seq}[T]$, and an operator $\# : \mathsf{Seq}[T] \to \mathsf{Nat}$ for sequence length, using one equation for each constructor case:

$$\begin{array}{ll} \varepsilon/S = \varepsilon & \#\varepsilon = 0 \\ (t;x)/S = \textbf{if } x \in S \textbf{ then } (t/S);x \textbf{ else } t/S & \#(t;x) = \#t + 1 \end{array}$$

2.1. Semantics

Let Ob be an unbounded set of object identifiers. Let Data be a set of data values, including Ob. In this paper, we conventionally let $o_1, o_2 \in \mathsf{Ob}$. A *communication event* has the form

$$msg \textbf{ from } o_1 \textbf{ to } o_2$$

where *msg* consists of Data. This event is considered an *output event* of o_1 and an *input event* of o_2. For observable events, o_1 and o_2 are distinct. The sets of observable input and output events of an object o are denoted IN_o and OUT_o, respectively, and are by definition disjoint. Their union is denoted $INOUT_o$.

An *alphabet* for an object o is a subset of $INOUT_o$. An alphabet of o may cover certain aspects of the communication of o. In the next section we introduce syntax for statically defined alphabets. A *trace set* $\mathcal{T}_\alpha \subseteq \mathsf{Seq}[\alpha]$ is a prefix-closed set of well-formed sequences.

Definition 2.1. A *specification* Γ is a triple $\langle o, \alpha, \mathcal{T} \rangle$ where (1) $o \in \mathsf{Ob}$ is an object identifier, (2) α is a possibly infinite alphabet for o, and (3) $\mathcal{T}$ is a trace set over α.

For any specification Γ, we can derive a *communication environment* $\mathcal{E}(\Gamma)$ of objects communicating with the object of Γ. In an ODS setting, we generally think of the communication environment as unbounded. Since the specification Γ does not need to cover all aspects of the behavior of o, we say that Γ is an *interface specification* (of o).

In the following we consider object-oriented distributed systems where communication is achieved through remote methods calls. In order to achieve asynchronous communication, we model a method call through two events: the event representing the initiation of a call, and the event representing its completion. Let Mtd be an unbounded set of method names, and let $m \in \mathsf{Mtd}$. For a call by o_1 to method m of o_2, the initiation event is generated by the caller o_1 and is represented by $invoc(m(\mathrm{E}))$ **from** o_1 **to** o_2, and the completion event is generated by the callee o_2 and represented by $comp(m(\mathrm{E}'))$ **from** o_2 **to** o_1 where E and E′ denote lists of data. In order to increase readability, we represent these events by $o_1 \rightarrow o_2.m(\mathrm{E})$ and $o_1 \leftarrow o_2.m(\mathrm{E}')$ respectively. By abstracting from the caller and the actual parameters, the sets of invocation and completion events of a method m in a callee o_2 from different callers and with different actual parameters are denoted $\rightarrow o_2.m$ and $\leftarrow o_2.m$, respectively.

As we consider asynchronously communicating objects, a caller may communicate while (passively) waiting for a completion and a callee may communicate while performing a method. Consequently, other events can be observed in between the initiation and completion of any given call. However, when we consider the history of observable behavior every completion event must be preceded by a *corresponding invocation*, which gives rise to the following notion of well-formedness for communication histories:

$$\begin{aligned} wf(\varepsilon) &= true \\ wf(t;(o{\rightarrow}o'.m(\mathrm{E}))) &= wf(t) \\ wf(t;(o{\leftarrow}o'.m(\mathrm{E}))) &= wf(t) \wedge \#(t/o{\rightarrow}o'.m) > \#(t/o{\leftarrow}o'.m) \end{aligned}$$

where $\#(t/o{\rightarrow}o'.m)$ is the length of the trace t restricted to invocation events of the method m from o to o', and similarly for completion events. Refinement for partial specifications can be defined as follows:

Definition 2.2. A specification $\langle o, \alpha, \mathcal{T} \rangle$ of an object o *refines* another specification $\langle o, \alpha', \mathcal{T}' \rangle$ of o if $\alpha' \subseteq \alpha$ and $\forall t \in \mathcal{T} \cdot t/\alpha' \in \mathcal{T}'$.

This notion of refinement corresponds to the subset relation on projected trace sets in the sense that if $\langle o, \alpha, \mathcal{T} \rangle$ refines $\langle o, \alpha', \mathcal{T}' \rangle$ then $\{t/\alpha' \mid t \in \mathcal{T}\} \subseteq \mathcal{T}'$. Note that a specification may refine several specifications with (partially) disjoint alphabets. The composition of specifications may be introduced to define partial components or system aspects in the sense of distributed services [18, 19].

2.2. Syntax

Interface specifications may be given in a generic manner. We shall refer to such generic specifications as *behavioral interfaces*. An object may support a number of interfaces. As Maude does not provide a syntax for specification of observable behavior, statically defined alphabets, nor methods (not even in Full Maude [8]), we introduce a syntax for observable behavior by means of object-oriented interfaces. The syntax for behavioral interfaces is given in Figure 1 and explained below.

```
interface F (⟨context parameters⟩)
  inherits F_1, F_2, ..., F_i
begin
with cointerface
    op m_1(...)
      ...
    op m_n(...)
  spec <predicate on the local trace>
  where <auxiliary function definitions>
end
```

Figure 1. A syntax for behavioral interfaces.

An interface can have context parameters, which typically describe its minimal required environment, representing static links needed by objects that support the interface. An initiation and a completion event is associated with each method declaration in the interface (ranging over method parameters). In specification formulas, the keyword "*this*" represents the object supporting the interface.

Mutual dependency. Let objects be typed by interfaces. By requiring in an interface I that the caller supports a specific type, a so-called *cointerface*, we restrict the objects that may call the methods of the interface I, while allowing *this* object to call cointerface methods. This opens up for interaction with a caller during the execution of a method. In an implementation language, access to the *caller* may be provided by an explicit parameter as in Maude, or implicitly as in Creol [22]. Cointerfaces give strong typing in an asynchronous setting. Semantically a cointerface declaration augments the alphabet of the interface, as events related to the cointerface methods are added.

Inheritance. Multiple inheritance is allowed for interfaces, but cyclic inheritance graphs are not allowed. In a subinterface, additional methods and behavioral constraints can be declared. A cointerface restriction applies to the locally declared methods; i.e., in Figure 1 the cointerface restriction applies to methods $m_1, \ldots, m_n$ but not to method declared in interfaces $F_1, \ldots, F_i$. If an interface F is declared with an inheritance clause, the alphabets of the super-interfaces are included in the alphabet of F.

Definition 2.3. The *interface alphabet* of an object o with respect to an interface F, denoted $\alpha_{o:F}$, is defined as the set of events of the forms

1. $invoc(m(\mathrm{E}))$ **from** o' **to** o, and $comp(m(\mathrm{E}'))$ **from** o **to** o', for m declared in F with parameter types such that E and E' are type correct input and output parameters, respectively
2. $invoc(m(\mathrm{E}))$ **from** o **to** o', and $comp(m(\mathrm{E}'))$ **from** o' **to** o, for m declared in (or inherited by) the cointerface, with parameter types such that E and E' are type correct input and output parameters
3. any event in $\alpha_{o:F'}$ where F' is a super-interface of F.

An interface has a specification predicate which captures the requirement of the interface to the communication history of objects supporting the interface, when restricted to the alphabet of events associated with the interface.

Definition 2.4. Let $F, F_1, \ldots, F_n$ be interfaces with corresponding specification predicates $P, P_1, \ldots, P_n$ and let h range over histories over the alphabet $\alpha_{this:F}$. If F inherits $F_1, \ldots, F_n$, then the *interface specification* of F is given as the conjunction $P(h) \wedge P_1(h/\alpha_{this:F_1}) \wedge \ldots \wedge P_n(h/\alpha_{this:F_n})$.

Assume-guarantee predicates. In ODS, the environment in which an object exists is subject to change, and specifications are relative to an assumed behavior of the environment. We adapt the assume-guarantee specification style [23, 24, 29] to the setting of observable behavior. Assumptions should express restrictions on the inputs and guarantees should express restrictions on the outputs. However in the context of interaction it is often difficult to formulate assumptions and guarantees separately, since requirements to outputs may depend on previous input, and requirements to inputs may depend on previous output. Instead we use a single predicate P which relates input and output events, and extract an assumption part and a guarantee part from P.

Definition 2.5. Let *IN* and *OUT* denote the sets of input and output events for *this* interface. An *assume-guarantee* predicate is derived from the specification **spec** $P(h)$, where the assumption part A and the guarantee part G are defined by the following equations:

$$\begin{aligned} A(\varepsilon) &= \mathit{true} \\ A(h;x) &= A(h) \wedge ((x \in \mathit{IN} \wedge P(h)) \Rightarrow P(h;x)) \\ G(\varepsilon) &= \mathit{true} \\ G(h;x) &= G(h) \wedge (A(h;x) \Rightarrow P(h;x)) \end{aligned}$$

The trace set given by the specification **spec** $P(h)$ is $\{h \mid G(h)\}$.

Note that the trace sets $\{h \mid G(h)\}$ and $\{h \mid A(h)\}$ are both prefix-closed, and that their intersection is the largest (prefix-closed) trace set contained in $\{h \mid P(h)\}$. For inherited interfaces, trace sets are inherited by intersection when restricted to the relevant alphabets of the respective super-interfaces. Thus, an interface will always refine its super-interfaces.

Assumptions are the responsibility of the objects in the environment. The assumption part ensures that each input is acceptable for the object supporting the interface, assuming no previous violation. Guarantees are the responsibility of the object supporting the interface; they are guaranteed when the assumption holds. Thus, the guarantee part ensures that each output is acceptable, assuming the assumption holds. It follows from the refinement notion given above that an actual environment is required to refine the trace set given by A, and an implementation of the interface is required to refine the trace set given by G.

2.3. Example: A Minimal Interface

Behavioral interfaces are illustrated through a distributed system for resource sharing, where nodes in a network have an initial amount of local resources and may borrow from each other in order to perform required tasks. A node may only lend its own resources, borrowed resources may only be returned. The nodes have two interfaces: one for borrowing from and one for returning resources to each other, and one interface allowing users to put resource requests on the nodes and to release the nodes. One may add a third interface specifying the actual tasks to be done on the network; however, our focus here is on semantic assumptions and guarantees concerning the borrowing and returning of resources, so we do

not need to specify the actual tasks. Strong typing and cointerfaces guarantee that only nodes may call *borrow* methods and thus *return* call-backs will be understood.

```
interface Client(init:Nat)
begin
with Client
  op borrow(a:Nat out b:Nat)
  op return(a:Nat)
  ⟨specification⟩
end
```

```
interface UserIF
begin
with User
  op acquire(a:Nat)
  op release
end
```

The parameter *init* on a client specifies its initial amount of resources. Denote by *caller* an arbitrary *Client* object in the environment of *this Client* object, as required by the cointerface. The alphabet of *Client* is given by the following events (for $n : Nat$):

Invocation	Completion	Comment
$caller \rightarrow this.borrow(n)$	$caller \leftarrow this.borrow(n)$	lend n resource units
$this \rightarrow caller.borrow(n)$	$this \leftarrow caller.borrow(n)$	ask for n resource units
$caller \rightarrow this.return(n)$	$caller \leftarrow this.return$	recover n resource units
$this \rightarrow caller.return(n)$	$this \leftarrow caller.return$	return n resource units

As the *return* method has no out-parameters, *return* completions will have no parameters. We define the following specification predicate in the *Client* interface:

$$\begin{array}{ll}\textbf{spec} & 0 \leq lent(this,h) \leq init \wedge \forall c : Client \cdot lent(c,h) \geq 0\\ \textbf{where} & lent(o,h) = sum((h/ \leftarrow o.borrow).b) - sum((h/ \rightarrow o.return).a)\end{array}$$

where the predicate $lent(o,h)$ captures the amount of resources lent by an object o to other client objects, and the function *sum* computes the sum of a sequence of numbers. Dot notation is used to extract parameter values from a sequence; e.g., $(h/ \leftarrow this.borrow).b$ denotes the sequence of values of the b parameter of the subsequence of borrow completions in h.

The specification states that a given client may not lend out more resources than it got initially, and for every other client in the system it may not return more resources to a client than it has borrowed from that client. Thus, resources borrowed from a client may neither be lent out nor returned to other clients.

By considering the different kinds of events, the assumption part of the specification reduces to

$$\begin{array}{lll}A(\varepsilon) & = true & \\ A(h; o \rightarrow this.return(n)) & = A(h) \wedge lent(this,h) \geq n & \\ A(h;x) & = A(h) & otherwise\end{array}$$

The assumption predicate states that the environment (as a whole) may not return more resources than it has borrowed. The *otherwise* equation in a definition is applied only when a term does not match the left hand side of any of the other equations. In the definition of A above, the *otherwise* equation applies when the history ends with any output event as well as with the input events $o \rightarrow this.borrow(n)$ and $this \leftarrow o.return$, which are not restricted by the *Client* specification. The specification holds for the input

event $this \leftarrow o.borrow(n)$ since $lent(this,h;this \leftarrow o.borrow(n)) \geq 0$ follows from $lent(this,h) \geq 0$. In the final case of an event x to consider, $o \rightarrow this.return(n)$, the only non-trivial part of the *Client* specification is $lent(this,h;x) \geq 0$, which is captured by the second equation above.

The guarantee part of the specification reduces to

$$\begin{array}{lll} G(\varepsilon) & = true & \\ G(h;o \leftarrow this.borrow(n)) & = G(h) \wedge (A(h) \Rightarrow lent(this,h) \leq init - n) & \\ G(h;this \rightarrow o.return(n)) & = G(h) \wedge (A(h) \Rightarrow lent(o,h) \geq n) & \\ G(h;x) & = G(h) & otherwise \end{array}$$

The guarantee predicate expresses that *this* object does not lend out more local resources than it has, and that it cannot return to any client more resources than it has borrowed from that client. The *otherwise* equation covers any input event x, for which $A(h;x)$ implies the *Client* specification, as well as the events $this \rightarrow o.borrow(n)$ and $o \leftarrow this.return$, which are not restricted by the *Client* specification.

The two interfaces above are connected by introducing an interface *Node*, inheriting both *Client* and *UserIF*. This results in client-like nodes controllable by *Users* which may request that a node acquires a certain resource power (by borrowing from others when needed) and by releasing the node (so that it may return borrowed resources).

interface *Node*(*init* : *Nat*)
  inherits *Client*(init), *UserIF*
begin
end

Thus, the alphabet of *Node* is the union of the alphabets of the *Client* and *UserIF* interfaces, and *Node* refines both of the inherited interfaces.

3. Rewriting Logic and Maude

This section gives a brief introduction to rewriting logic [25] and Maude [8]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Rewrite rules apply to fragments of a state configuration. If rewrite rules may be applied to non-overlapping fragments of the configuration, the transitions may be performed in parallel. Consequently, rewriting logic (RL) is a logic which easily captures concurrent change. A number of concurrency models have been successfully represented in RL [8, 25], including Petri nets, CCS, Actors, and Unity.

Informally, a state configuration in RL is a multiset of terms of given types, specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [14] style. Memberships express that a term belongs to a given sort. When modeling systems, configurations may include system components modeled by terms of the different types defined in the equational logic. An RL object is a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [8].

RL extends algebraic specification techniques with rewrite rules to capture the dynamic behavior of a system, supplementing the equations defining the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the equations of E. Rewrite rules may have a condition (a conjunction of rewrites, equations, and memberships) which must hold for the main rule to apply. Each rule describes how a part of a configuration can evolve in one transition step:

$$\textbf{rl}\ [\text{label}] : \mathit{subconfiguration} \longrightarrow \mathit{subconfiguration}$$
$$\textbf{crl}\ [\text{label}] : \mathit{subconfiguration} \longrightarrow \mathit{subconfiguration}\ \textbf{if}\ \mathit{condition}$$

An unconditional rule with an *if-then-else* expression as the right hand side may alternatively be given as two complementary conditional rules. Rules in RL may be formulated at a high level of abstraction, closely resembling a structural operational semantics [27], as argued in [26]. The Maude system supports analysis of RL specifications.

3.1. Example: Implementation of the Nodes

We present an executable Maude specification which implements the *Node* interface specification given in Section 2.3. A node object is defined as an RL object $\langle O : \mathit{Node} \mid \mathit{local} : \_, \mathit{brwd} : \_, \mathit{state} : \_ \rangle$, where O ranges over object identifiers. The attribute *local* indicates the amount of local resources the node has available. The attribute *brwd* records the amount of resources *this* object has borrowed from its clients, thus indicating the knowledge the object has of its environment. Technically, the attribute is a list of pairs which contain the identities of known clients in the environment and the amount of resources borrowed from each of these clients. The attribute *state*, with possible values *free*, *pre*, *acq*, indicates a lock on lending: If this object is in the process of acquiring resources it is in the state *pre*, and after a completion of acquire it is in the state *acq*. In both cases it will not lend resources to other clients. The client interacts asynchronously with its environment by means of message passing. The Maude implementation is given as a set of rewrite rules on (sub)configurations in Figure 2. Configurations, which capture system states, are defined as multisets of objects and messages. Pattern matching is done modulo associativity, commutativity, and identity (with *none* as identity element) for the multiset combinator, denoted by whitespace as conventional in Maude, and modulo associativity and identity (with ε as identity element) for the list combinator, denoted by $+$. Method names are written as quoted identifiers (as in *'borrow*).

In addition to the rewrite rules in Figure 2, the operators $+$ and *bwd* are specified by the equations

$$\begin{aligned} (X,N) + B + (X,M) &= (X, N+M) + B \\ \mathit{bwd}(\varepsilon) &= 0 \\ \mathit{bwd}(B + (X,N)) &= \mathit{bwd}(B) + N \end{aligned}$$

Consequently, terms with these operators are reduced in between rewrite steps. The associative operator $+$ is used to construct lists of (*Ob*,*Nat*) pairs, *bwd* computes the sum of borrowed resources. The predefined functions *min* and *max* return the minimum and maximum of two natural numbers, respectively. The rule *return1* generates a *return* invocation to a client from which the node has borrowed resources when the state of the node is free. In this rule, the amount of borrowed resources from that client is reset to 0 in the *brwd* list, thus retaining the node's knowledge of the environment. To keep the Maude specification simple, the specification allows an execution sequence in which the rule *acquire1* is applied repeatedly once it is applicable (even if the rule *acquire2* has become applicable as well).

rl [borrow] : $(\mathit{invoc}\ {}'\mathit{borrow}(M)\ \mathit{from}\ O'\ \mathit{to}\ O)\ \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : \mathit{free}\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : \mathit{max}(0, N-M), \mathit{brwd} : B+(O',0), \mathit{state} : \mathit{free}\rangle$
$(\mathit{comp}\ {}'\mathit{borrow}(\mathit{min}(N,M))\ \mathit{from}\ O\ \mathit{to}\ O')$.

crl [noborrow] : $(\mathit{invoc}\ {}'\mathit{borrow}(M)\ \mathit{from}\ O'\ \mathit{to}\ O)\ \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : C\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B+(O',0), \mathit{state} : C\rangle\ (\mathit{comp}\ {}'\mathit{borrow}(0)\ \mathit{from}\ O\ \mathit{to}\ O')$ **if** $C \neq \mathit{free}$.

rl [compborrow] : $(\mathit{comp}\ {}'\mathit{borrow}(M)\ \mathit{from}\ O'\ \mathit{to}\ O)\ \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : C\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B+(O',M), \mathit{state} : C\rangle$.

crl [return1] : $\langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B+(X,M)+B', \mathit{state} : \mathit{free}\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B+(X,0)+B', \mathit{state} : \mathit{free}\rangle\ (\mathit{invoc}\ {}'\mathit{return}(M)\ \mathit{from}\ O\ \mathit{to}\ X)$ **if** $M > 0$.

rl [return2] : $(\mathit{invoc}\ {}'\mathit{return}(M)\ \mathit{from}\ O'\ \mathit{to}\ O)\ \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : C\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : N+M, \mathit{brwd} : B, \mathit{state} : C\rangle$.

rl [release] : $(\mathit{invoc}\ {}'\mathit{release}(\varepsilon)\ \mathit{from}\ O'\ \mathit{to}\ O)\ \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : C\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : \mathit{free}\rangle\ (\mathit{comp}\ {}'\mathit{release}(\varepsilon)\ \mathit{from}\ O\ \mathit{to}\ O')$.

crl [acquire1] : $(\mathit{invoc}\ {}'\mathit{acquire}(M)\ \mathit{from}\ O'\ \mathit{to}\ O)\ \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : (X,N')+B, \mathit{state} : C\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B+(X,N'), \mathit{state} : \mathit{pre}\rangle\ (\mathit{invoc}\ {}'\mathit{acquire}(M)\ \mathit{from}\ O'\ \mathit{to}\ O)$
$(\mathit{invoc}\ {}'\mathit{borrow}(M-(N+N'+\mathit{bwd}(B)))\ \mathit{from}\ O\ \mathit{to}\ X)$ **if** $M > (N+N'+\mathit{bwd}(B)) \wedge C \neq \mathit{acq}$.

crl [acquire2] : $(\mathit{invoc}\ {}'\mathit{acquire}(M)\ \mathit{from}\ O'\ \mathit{to}\ O)\langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : C\rangle$
$\longrightarrow \langle O : \mathit{Node} \mid \mathit{local} : N, \mathit{brwd} : B, \mathit{state} : \mathit{acq}\rangle\ (\mathit{comp}\ {}'\mathit{acquire}(\varepsilon)\ \mathit{from}\ O\ \mathit{to}\ O')$
if $M <= N + \mathit{bwd}(B) \wedge C \neq \mathit{acq}$.

Figure 2. Rewrite rules capturing node behavior. Maude variables are written as uppercase letters.

3.2. Reflection and The Maude Metalevel

Rewriting logic is reflective in the sense that there is a finitely presented rewrite theory $\mathcal{U}$ which is *universal*: any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) can be represented in $\mathcal{U}$. Let C and C' be configurations and $\mathcal{R}$ be a set of rewrite rules. We write $\mathcal{R} \vdash C \rightarrow C'$ to express that C may be rewritten to C' in the rewrite theory $\mathcal{R}$. Informally, a configuration C and the set $\mathcal{R}$ of rewrite rules of a specification in RL may be uniformly represented by terms $\overline{C}$ and $\overline{\mathcal{R}}$ at the metalevel. Using this notation, we have the equivalence [7]

$$\mathcal{R} \vdash C \rightarrow C' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{C}\rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{C'}\rangle,$$

which states that if a term C in the rewrite theory $\mathcal{R}$ can be rewritten to a term C', then the meta-representation of C in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C}\rangle$ can be rewritten to the meta-representation of C' in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C'}\rangle$, in the universal rewrite theory $\mathcal{U}$, and vice versa. Maude includes facilities to meta-represent a rewrite theory $\mathcal{R}$ and to apply rules from $\mathcal{R}$ to the meta-representation of a term C by so-called *descent functions*.

Metalevel rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by defining an interpreter function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Metalevel rewrite rules may further be used to modify

```
crl [exec] :
  ⟨M : MetaRep | curTerm : T, curModule : MOD, labels : L LS, failedRules : FR⟩
  ⟨History : H⟩                    ⟶
  if RES :: Result4Tuple then
    ⟨M : MetaRep | curTerm : getTerm(RES),curModule : MOD, labels : LS L, failedRules : ε⟩
    ⟨History : H ; getNewMessages(T, getTerm(RES), MOD, H)⟩
  else
    ⟨M : MetaRep | curTerm : T, curModule : MOD, labels : LS L, failedRules : FR L⟩
    ⟨History : H⟩ fi
if  RES := metaXapply([MOD], T, L, none, 0, unbounded, 0) ∧ #FR ≤ #LS.
```

Figure 3. The metalevel rewrite strategy $\mathcal{S}_{monitor}$ records the communication history. The membership expression *RES :: Result4Tuple* states that the rewrite bound to *RES* succeeds, using a condition of the form *RES* := *term* to bind a term to *RES*.

a configuration or the rule set of a rewrite theory. Hence, metalevel rewriting can be used as a wrapper around a rewrite theory $\mathcal{R}$ in order to abstractly mimic a more elaborate rewrite theory $\mathcal{R}'$ extending $\mathcal{R}$. Further details on the theory and the use of reflection in RL and Maude may be found in [7, 8, 9]. Metalevel rewriting is exploited in Sections 4 and 5 below.

4. Monitoring and Testing Executable Models

The observable behavior of an executable model can be monitored by recording the *communication history* from an execution of the model: This can be done *transparently* with the aid of the Maude metalevel without modifying the original specification. We can further test that the execution conforms to the behavioral specification of the model by defining metalevel predicates that operate on the recorded history and block execution if a violation occurs.

To execute a specification at the metalevel, we develop a custom *strategy*; i.e., rewrite rules which apply to the meta-representation of the model. Thus the current state may be inspected in-between rewrites. This enables us to record a communication history while executing a specification: We can check whether the application of a rewrite rule results in the emission of a new message by comparing the metalevel representations of the configuration before and after the rule application.

The object $\langle M : MetaRep \mid curTerm : \_, curModule : \_, labels : \_, failedRules : \_ \rangle$ is used to store the information needed to control consecutive metalevel rewrites. The attribute *curTerm* contains the meta-representation of the current object level configuration, *curModule* is the meta-representation of the name of the object level module in which the rewrites will be performed, *labels* is a list of rule labels from this module, and *failedRules* contains a list of labels for rules that are not applicable to *curTerm*.

The object $\langle History : \_ \rangle$ has an attribute h which contains the actual communication history recorded at runtime as a message list. This object is distinct from the objects at the object level and is consequently not modified by nor needed for the application of any rewrite rule from the object level specification.

The custom strategy $\mathcal{S}_{monitor}$ is implemented as a conditional rewrite rule $exec : MetaRep \times History \rightarrow MetaRep \times History$ (see Figure 3). The actual rewriting is done by the built-in Maude function *metaXapply*, which returns a tuple from which the rewritten term is obtained using *getTerm*. Note that whitespace in Maude denotes list concatenation: If L is a label and LS is a list of labels, then $L\ LS$ is a non-empty list

of labels. The strategy $\mathcal{S}_{monitor}$ applies rules from the *labels* list to the metalevel configuration in *curTerm* in a round-robin fashion. (A position-fair strategy for random rule selection based on a pseudo-random number generator is given in [20].) If no rule is applicable, the execution will terminate. An auxiliary function *getNewMessages* compares the term T to the new system configuration; i.e., the result of applying the rule labeled L to T. If there are new communication messages in the new configuration, the attribute h of the history object is extended with the new messages. If there are several new messages, these are caused by concurrent actions and may therefore be added to the history in an arbitrary order.

The custom strategy $\mathcal{S}_{test}$ is defined by extending $\mathcal{S}_{monitor}$ with functionality to check whether a given rule application will lead to an illegal state, as specified by a predicate parameter. We consider predicates on communication histories as defined by behavioral interfaces. To obtain a compositional system, the predicate on the global history will be formulated as the conjunction of the requirement specifications of a number of behavioral interfaces, possibly associated with different objects. Behavioral specifications for specific objects are represented by predicates on the global history, restricted to an appropriate subset of possible communication events; i.e., the global system should refine the behavioral specifications of its objects.

The $\mathcal{S}_{test}$ strategy blocks further execution once the system attempts to reach an illegal state violating the predicate on the global history. To test a particular object o against a behavioral specification $\langle o, \alpha, \mathcal{T}_\alpha \rangle$, the testing predicate can be expressed as $P(h) = h/\alpha \in \mathcal{T}_\alpha$. For behavioral requirements given as a predicate $P : \mathsf{Seq}[\alpha] \to \mathsf{Bool}$, defined by a convergent set of equations, membership in the trace set is effectively computable by reducing $P(h/\alpha)$ for the current global history h.

The $\mathcal{S}_{test}$ strategy is implemented in Maude by extending the conditional *exec* rule with a branch which checks the given predicate for the recorded communication history between each rewrite step, and blocks execution if the predicate is violated. A Maude function *CheckPredicate* : *Pred* × *MsgList* → *Bool* is used for this purpose. A predicate is specified using a constant H as a placeholder for the actual communication history. At run-time *CheckPredicate* parses the predicate specification against the actual history, calls any auxiliary predicates, and returns a boolean value indicating whether the history after the rewrite step would be in compliance with the predicate or not. If the execution is blocked by the strategy, the recorded history provides an error trace for the system run, describing how the specification was violated.

Example. The acceptable behavior of a node X which behaves according to the *Client* interface (introduced in Section 2.3) can be expressed by a Maude operator *AccBeh*

eq *AccBeh*(ε) = *true*
eq *AccBeh(H ; MSG from X to Y)* = *P(H/X ; MSG from X to Y)*

where P is the specification predicate of the *Client* interface (replacing the placeholder symbol *this* by X), and where the notation H/X abbreviates the projection $H/INOUT_X$. Since P in the Maude specification above is a *global* predicate that spans all objects, there is no need to pass the object identifier as a separate parameter to *AccBeh*. In addition, since *AccBeh* is checked for each input and output event incrementally, we need not check all of P every time the history is extended. It suffices to test the incremental guarantee part $G'(h;x) = x \in OUT_X \wedge P(h) \Rightarrow P(h;x)$. Using the results of Section 2.3, this incremental guarantee predicate reduces to

$$
\begin{array}{lll}
G'(h; o_1 \leftarrow o_2.borrow(n)) & = lent(o_2,h) \leq init - n & \\
G'(h; o_2 \rightarrow o_1.return(n)) & = lent(o_1,h) \geq n & \\
G'(h;x) & = true & otherwise
\end{array}
$$

5. Simulation of Open Environments for Testing

An open environment can be *simulated* such that the behavior of abstract objects is exclusively defined by the behavioral interfaces. Interface assumptions on the observable behavior may be used to generate arbitrary environment behavior within the limits imposed by the assumption predicate.

5.1. Syntactic Simulation of Open Environments

At the object level, a rewrite theory is used to syntactically simulate the unknown environment. In an open environment, objects may be created and destroyed dynamically during execution. To mimic the open environment, we define a term containing a set *absIDs* of (abstract) object identifiers representing objects which may currently interact with the system: $\langle E : Envir \mid absIDs : \_, sysIDs : \_, seed : \_ \rangle$. The set *absIDs* will be used to generate input messages to the objects of the system. Real system objects are represented as a set *sysIDs* of pairs $Obj \times Set[Mtds]$ which consist of object identifiers and sets of invocation and completion definitions corresponding to the input alphabets of the object's interfaces. The messages emitted by abstract objects are input to the real objects of the system. The *seed* attribute is used for message generation.

In order to produce arbitrary but syntactically correct input to the system from objects in the environment, we need to select an object o from *sysIDs* and produce a message to o (either calling a method available in the interface of o or replying to a call from o found in the history). For this purpose, we use a pseudo-random number generator [20] and let the function $next : Nat \rightarrow Nat$ produce new seed values for the environment. Let the function $genMsg : Obj \times Obj \times Set[Msg] \times Nat \rightarrow Msg$ generate a new message *msg* to an object o with alphabet α in the system from an object in the environment, such that $msg \in \alpha$. In case of parameterized messages, type correct parameter values must be generated. For bounded value-sets, this is feasible by representing the sets explicitly and choosing an element by means of the pseudo-random generator. For unbounded numbers, the pseudo-random generator may be applied directly without explicit representation. For other unbounded value-sets, bounded subsets may be useful for test purposes. More advanced solutions may be devised, but are beyond the scope of this paper.

The rewrite rule for message generation is given by:

$$
\begin{array}{l}
\textbf{rl}\ [\text{msg-gen}] : \langle E : Envir \mid absIDs : o_1\, A, sysIDs : (o_2, \alpha)\, C, seed : X \rangle \\
\longrightarrow \langle E : Envir \mid absIDs : o_1\, A, sysIDs : (o_2, \alpha)\, C, seed : next(X) \rangle\ genMsg(o_1, o_2, \alpha, X)
\end{array}
$$

5.2. Semantic Simulation of Open Environments for Testing

At the metalevel, a rewrite theory is used to semantically simulate the unknown environment. Minimal behavioral requirements for open environments are given by assumptions in the system interfaces. Define a metalevel strategy $\mathcal{S}_{restrict}$ which *restricts* a rewrite system to behave according to a predicate on observable behavior. This strategy is similar to $\mathcal{S}_{test}$, but where $\mathcal{S}_{test}$ halts the execution when the application

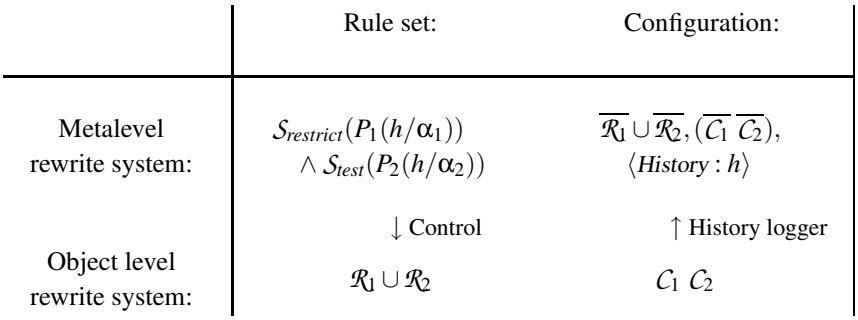

Figure 4. Reflective testing of observable behavior in open environments.

of an enabled rule would violate the predicate, $\mathcal{S}_{restrict}$ tries another enabled rule from the *labels* list of the *MetaRep* object instead. Open environments do not terminate; if no rewrite rule is applicable to any position of *curTerm*, the strategy changes the seed value and retries the rules.

The abstract environment specification can now be used as a *testbed* for an actual programmed component (see Figure 4). Let $\mathcal{R}_1$ be an object level set of rewrite rules generating (and possibly garbage collecting) messages. Rules from $\mathcal{R}_1$ may be applied to a configuration $\mathcal{C}_1$ consisting of an *Envir* object. Let $\mathcal{R}_2$ be the object level set of rewrite rules applicable to the concrete objects in a configuration $\mathcal{C}_2$, e.g., the given component, with synchronization constraints on the internal state. Let α_1 and α_2 be alphabets associated with the objects of $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, such that $\alpha_1 \subseteq \alpha_2$. Let P_1 and P_2 be predicates observationally specifying the environment and actual component, respectively. If several interfaces are considered, P_1 will be the conjunction of assumptions and P_2 the conjunction of guarantees, restricted to the relevant alphabets. The metalevel strategy $\mathcal{S}_{restrict}$ restricts rule application from $\mathcal{R}_1$ to acceptable environment behavior, providing an abstract, open environment which may behave in any way that does not violate the predicate P_1. We here combine two metalevel strategies which react differently to the violation of predicates: $\mathcal{S}_{restrict}$ will restrict rule application so that the communication history conforms to the predicate, and $\mathcal{S}_{test}$ will halt the execution and produce an error object if the predicate does not hold. By specifying one predicate that spans only messages from the objects of the component, and one that spans all objects, and executing the former with $\mathcal{S}_{test}$ and the latter with $\mathcal{S}_{restrict}$, we can test whether the programmed component executes correctly provided that the environment does so.

5.3. Testing the Node Example in an Open Environment

This test scenario was implemented in Maude by a metalevel rewrite rule *exec-test* similar to the rule in Figure 3, combining the $\mathcal{S}_{restrict}$ and $\mathcal{S}_{test}$ strategies described above. The metalevel specification was used to test the Node implementation described in Section 3.1. One concrete Node object was tested in an environment of 4 abstract nodes, simulated as described in Section 5.1. The rewrite rules for node behavior (Figure 2) were compared to the *Client* interface specification (Section 2.3) using $\mathcal{S}_{test}$, whereas application of the *msg-gen* rule was restricted by $\mathcal{S}_{restrict}$ to conform to the increment of the assumption A_{Client}; i.e., $A'(h;x) = x \in IN_X \wedge P(h) \Rightarrow P(h;x)$. By the results of Section 2.3, this reduces to

$$\begin{aligned} A'(h; o_1 \rightarrow o_2.return(n)) &= sum((h/ \rightarrow o_2 return).a) + n \leq sum((h/ \leftarrow o_2.borrow).b) \\ A'(h;x) &= true \qquad otherwise \end{aligned}$$

The specification was executed a number of times with different initial seed values for the message generation rules. Type correct parameter values for the messages were generated by randomly choosing elements from a predefined value-set using the pseudo-random generator.

The tests suggest that the simulated environment conforms to the assumption part of the specification, due to the restrict strategy, and that a given concrete node implemented by the rules in Figure 2 fulfills the guarantee part. The specification is not deadlock-free; e.g., two nodes trying to borrow from each other may deadlock. However, this cannot be detected with a single test object, since completions of pending *borrow* invocations are eventually generated by the environment if a fair rewrite strategy is used.

When applying the *exec-test* rule to this non-terminating specification 2000 times for one particular simulation, the result (after 44991660 rewrites) was a trace of 326 events involving the concrete object. According to the tests, the model implements the specification. When adding rules which violated the guarantee, this violation was detected. Thus the example suggests that the approach works well for an initial model validation, although coverage estimates for the approach are beyond the scope of this paper.

6. Related and Future Work

We do not attempt to fully survey the extensive literature on monitoring and testing here. Object-oriented systems are usually tested by assuming that the implementation details are known [6]. Many previous history-based [11, 28, 32] and automata-based [4, 31, 33] approaches require specific and deterministic test cases to be defined. In contrast we propose *adaptive testing* of components based on assume-guarantee specifications which capture open environments; i.e., the behavior of the environment is arbitrary within the constraints imposed by an assumption predicate. Thus, the strategy proposed in this paper adapts to the present state of the system. Adaptive testing is more cost efficient than random testing [5]. In our approach, the challenge is to model the environment such that all constraints are explicit in the assumption predicate rather than implicit in the syntactic model. For open environments, it remains an interesting challenge to improve adaptive testing by integrating reasonable test strategies based on, for example, uniformity or regularity hypotheses [13]; in particular, open environments are both nonterminating and highly nondeterministic. Although it is technically straightforward to differentiate between test cases by additional assumption predicates to further restrict the open environment, it remains future work to derive an appropriate notion of coverage for open environments from the use of such a technique. In *Credo*, we currently investigate testing techniques for Creol [22] using rewriting logic and Maude [8]. In this context, we plan to extend the approach of this paper with techniques based on fault injections [2].

Recently, the notion of full abstraction for concurrent objects has been studied in the context of trace semantics [1, 17]. These works propose a more fine-grained notion of observability than we have adopted in this paper. In particular, the propagation of object references in the environment and the creation of new class instances are discussed. A related generalization of the syntactic environment we have developed may improve its applicability to more refined adaptive testing strategies.

The specifications of observable behavior we have considered can easily be represented in rewriting logic. However, a more expressive specification language may be desirable. For example, our approach to open environment modeling could be combined with linear time temporal logic specifications on finite traces. An efficient algorithm in rewriting logic for verifying such formulas is given in [30].

Invariant-driven strategies for Maude similar to our $\mathcal{S}_{restrict}$ have recently been proposed in [12], but that paper considers predicates on states rather than observable behavior and does not consider the ap-

plication to open environments nor to testing. For open environments, random testing within the bounds of minimal assumptions seems more attractive than deterministic tests. Some experiments with socket extensions to Maude suggest that Maude processes may be used as demonstrated in this paper to simulate an open testing environment for actual components which communicate by means of a predefined set of messages with the Maude process via sockets. However, future work in this direction remains.

7. Conclusion

The main contribution of this paper is to describe an approach to the validation of black-box components in open environments by extending Maude models with a notion of observable behavior and develop execution strategies which exploit this extension. The paper shows how abstract specifications of open environments may be captured very naturally in a rewriting logic model extended with behavioral interfaces. The behavioral interfaces express safety requirements on the observable behavior of components. The approach is presented within a method-based object-oriented setting, but may easily be adjusted to general asynchronous message passing. Due to the reflective character of rewriting logic, supported by Maude, it is possible to define execution strategies at the metalevel. In this paper, we have used this facility in four ways. First, a strategy is defined to non-deterministically generate arbitrary input to a system. Second, a strategy is defined to transparently introduce the monitoring of a set of communication events. Third, a strategy is defined to restrict system input by semantic requirements on the observable behavior. Combining these strategies, the arbitrary behavior of open environments may be simulated within the bounds of minimal assumptions. The separation of object level and metalevel constraints facilitates experimenting with different assumptions on the environment. The same approach may also be used to execute a prototype model defined in terms of observable behavior, before deciding on its implementation details. Fourth, a strategy is defined to test whether an executable model is well-behaved with respect to semantic requirements on the observable behavior. Combining all four strategies, we obtain abstract validation environments for models of components or distributed applications, in which the environment is unspecified but subjected to minimal observational requirements. The approach is implemented and illustrated by application to an example of a peer-to-peer network where nodes may borrow resources from each other. Results from testing guarantees as well as assumptions are presented.

Acknowledgments. We are grateful to Eyvind W. Axelsen for contributing to the implementation of these ideas and to the anonymous referees for helpful comments.

References

[1] Ábrahám, E., Bonsangue, M. M., de Boer, F. S., Steffen, M.: Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes, in: *Proceedings of the 1st International Colloquium on Theoretical Aspects of Computing (ICTAC 2004)* (Z. Liu, K. Araki, Eds.), vol. 3407 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005, 37–51.

[2] Aichernig, B. K., Delgado, C. C.: From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems., in: *9th International Conference on Fundamental Approaches to Software Engineering (FASE 2006)* (L. Baresi, R. Heckel, Eds.), vol. 3922 of *Lecture Notes in Computer Science*, 2006, 324–338.

[3] Alpern, B., Schneider, F. B.: Defining Liveness, *Information Processing Letters*, **21**(4), October 1985, 181–185.

[4] Barbey, S., Buchs, D., Péraire, C.: A Theory of Specification-Based Testing for Object-Oriented Software, in: *Proceedings of the European Dependable Computing Conference (EDCC2)*, vol. 1150 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996, 303–320.

[5] Cai, K.-Y., Chen, T. Y., Li, Y.-C., Ning, W.-Y., Yu, Y. T.: Adaptive Testing of Software Components, *Proceedings of the Symposium on Applied Computing (SAC 2005)* (H. Haddad, L. M. Liebrock, A. Omicini, R. L. Wainwright, Eds.), ACM Press, 2005.

[6] Chen, H. Y., Tse, T. H., Chan, F. T., Chen, T. Y.: In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs, *ACM Transactions on Software Engineering and Methodology*, **7**(3), 1998, 250–295.

[7] Clavel, M.: *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*, CSLI Publications, Stanford, California, 2000.

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J. F.: Maude: Specification and Programming in Rewriting Logic, *Theoretical Computer Science*, **285**, August 2002, 187–243.

[9] Clavel, M., Meseguer, J.: Reflection in Conditional Rewriting Logic, *Theoretical Computer Science*, **285**, August 2002, 245–288.

[10] Dahl, O.-J.: Can Program Proving Be Made Practical?, in: *Les Fondements de la Programmation* (M. Amirchahy, D. Néel, Eds.), Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, December 1977, 57–114.

[11] Doong, R.-K., Frankl, P. G.: The ASTOOT Approach to Testing Object-Oriented Programs, *ACM Transactions on Software Engineering and Methodology*, **3**(2), 1994, 101–130.

[12] Durán, F., Roldán, M., Vallecillo, A.: Invariant-Driven Strategies for Maude, *Electronic Notes in Theoretical Computer Science*, **124**(2), 2005, 17–28, Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004).

[13] Gaudel, M.-C., James, P. R.: Testing Algebraic Data Types and Processes: A Unifying Theory, *Formal Aspects of Computing*, **10**(5–6), 1998, 436–451.

[14] Goguen, J. A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P.: Introducing OBJ, in: *Software Engineering with OBJ: Algebraic Specification in Action* (J. A. Goguen, G. Malcolm, Eds.), Advances in Formal Methods, chapter 1, Kluwer Academic Publishers, 2000, 3–167.

[15] Hoare, C. A. R.: *Communicating Sequential Processes*, International Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ., 1985.

[16] International Telecommunication Union: *Open Distributed Processing - Reference Model parts 1–4*, Technical report, ISO/IEC, Geneva, July 1995.

[17] Jeffrey, A., Rathke, J.: A Fully Abstract May Testing Semantics for Concurrent Objects, in: *Proceedings of the 17th Annual Symposium on Logic in Computer Science (LICS 2002)*, IEEE Computer Society Press, 2002, 101–112.

[18] Johnsen, E. B., Owe, O.: A Compositional Formalism for Object Viewpoints, in: *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)* (B. Jacobs, A. Rensink, Eds.), Klüwer Academic Publishers, March 2002, 45–60.

[19] Johnsen, E. B., Owe, O.: Object-Oriented Specification and Open Distributed Systems, in: *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl* (O. Owe, S. Krogdahl, T. Lyche, Eds.), vol. 2635 of *Lecture Notes in Computer Science*, Springer-Verlag, 2004, 137–164.

[20] Johnsen, E. B., Owe, O., Axelsen, E. W.: A Run-Time Environment for Concurrent Objects with Asynchronous Method Calls, *Electronic Notes in Theoretical Computer Science*, **117**(2), January 2005, 375–392, Proceedings of the 5th International Workshop on Rewriting Logic and its Applications (WRLA'04).

[21] Johnsen, E. B., Owe, O., Torjusen, A. B.: Validating Behavioral Component Interfaces in Rewriting Logic, *Electronic Notes in Theoretical Computer Science*, **159**, May 2006, 187–204, Proceedings of the IPM International Workshop on Foundations of Software Engineering (FSEN 2005).

[22] Johnsen, E. B., Owe, O., Yu, I. C.: Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems, *Theoretical Computer Science*, **365**(1–2), November 2006, 23–66.

[23] Jones, C. B.: *Development Methods for Computer Programmes Including a Notion of Interference*, Ph.D. Thesis, Oxford University, UK, June l981.

[24] Lamport, L.: *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley, 2002.

[25] Meseguer, J.: Conditional Rewriting Logic As a Unified Model of Concurrency, *Theoretical Computer Science*, **96**, 1992, 73–155.

[26] Meseguer, J., Roşu, G.: Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools, in: *Proceedings of the Second International Joint Conference on Automated Reasoning (IJCAR 2004)* (D. A. Basin, M. Rusinowitch, Eds.), vol. 3097 of *Lecture Notes in Computer Science*, Springer-Verlag, 2004, 1–44.

[27] Plotkin, G. D.: A Structural Approach to Operational Semantics, *Journal of Logic and Algebraic Progamming*, **60**, 2004, 17–139.

[28] Pretschner, A., Lötzbeyer, H., Philipps, J.: Model Based Testing in Incremental System Development, *Journal of Systems and Software*, **70**(3), 2004, 315–329.

[29] de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, Number 54 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, UK, November 2001.

[30] Roşu, G., Havelund, K.: Rewriting-Based Techniques for Runtime Verification, *Journal of Automated Software Engineering*, **12**(2), 2005, 151–197.

[31] Rusu, V., Marchand, H., Tschaen, V., Jéron, T., Jeannet, B.: From Safety Verification to Safety Testing, in: *16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004)* (R. Groz, R. M. Hierons, Eds.), vol. 2978 of *Lecture Notes in Computer Science*, Springer-Verlag, 2004, 160–176.

[32] Tyler, B., Soundarajan, N.: Black-Box Testing of Grey-Box Behavior, in: *3rd Intl. Workshop on Formal Approaches to Testing of Software (FATES 2003)* (A. Petrenko, A. Ulrich, Eds.), vol. 2931 of *Lecture Notes in Computer Science*, Springer-Verlag, 2004, 1–14.

[33] Van Aertryck, L., Benveniste, M., Le Metayer, D.: CASTING : A Formally Based Software Test Generation Method, *Proceedings of the 1st IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, IEEE Computer Society Press, November 1997.

[34] Venkatasubramanian, N., Talcott, C. L.: Reasoning About Meta Level Activities in Open Distributed Systems, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, August 1995.

[35] Walter, T., Schieferdecker, I., Grabowski, J.: Test Architectures for Distributed Systems - State of the Art and Beyond, in: *Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems (IWTCS'98)* (A. Petrenko, N. Yevtushenko, Eds.), vol. 131 of *IFIP Conference Proceedings*, Kluwer Academic Publishers, June 1998, 149–174.